14 Protocols and profiles using OAuth 2.0

This chapter covers

- User Managed Access (UMA), a protocol built on top of OAuth 2.0 for dynamic consent and policy management
- Health Relationship Trust (HEART), a profile of OAuth 2.0, OpenID Connect (OIDC), and UMA for healthcare-related scenarios
- International Government (iGov), a profile of OAuth 2.0 and OpenID Connect for government services

As you've seen by now, OAuth 2.0 is a powerful protocol, and it's good at what it does: delegation of access rights and communication of that authorization across HTTP. OAuth can't do many things on its own. If you need to go beyond what OAuth offers, it's a valuable tool in the toolbox, but it's not the only tool at your disposal. OAuth is a versatile building block in more complex systems.

In chapter 13, we looked at one major use case, user authentication, and a standard protocol built on top of OAuth that performs that function, OpenID Connect. In this chapter, we're going to look at several additional protocols and profiles of OAuth 2.0 that build advanced capabilities on this robust base. First we'll look at an application of OAuth that extends OAuth's capabilities to allow for user-to-user sharing and dynamic consent management. Then we'll look at a pair of efforts to profile OAuth and related protocols for specific domains, and how those efforts relate to the wider world. Note well: at the time of this writing, these specifications

are in flux and the final (or current) versions are likely to be a little different by the time you read this. We should also note that one of your authors is heavily involved in all three of these standardization and profiling efforts.

14.1 User Managed Access (UMA)

UMA[1] is a protocol built on top of OAuth 2.0 that allows the resource owner rich control over access to their resources through the use of an authorization server of the resource owner's choosing, either to software that they control or to software that another user may control. The UMA protocol allows for two main functions to be built on top of OAuth 2.0: user-to-user delegation, and handling of multiple authorization servers per resource server.

In other words, where OAuth 2.0 allows a resource owner to delegate to *client software* to act on their behalf, UMA allows a resource owner to delegate to *another user's client software* to act on that other user's behalf. Colloquially, OAuth enables Alice-to-Alice sharing (since Alice is running the client herself), whereas UMA enables Alice-to-Bob sharing. UMA additionally allows Alice to bring her own authorization server and introduce it to the resource server. Bob's client can discover Alice's authorization server once it tries to access Alice's resources.

UMA manages this trick by changing the relationships between traditional OAuth roles and defining a brand-new role in the process: the requesting party (RqP).[2] The resource owner manages the relationship between the authorization server and the resource server, setting up policies that allow third parties access to the resource. The client, in the control of the requesting party, can request an access token by presenting information about itself and about the requesting party in order to fulfill the requirements set by the resource owner. The resource owner doesn't interact with the client at all, and instead delegates access to the requesting party (figure 14.1).

Yes, the UMA dance is a fair shade more complex than the OAuth dance that we covered in chapter 2, but that's because it's solving a more complex problem. The protection API half of UMA is under the direction of the resource owner, whereas the authorization API half of UMA is under the direction of the requesting party. Each party and component has a part to play in the UMA dance.

14.1.1 Why UMA matters

Before we look in depth into how it works, why should you care about UMA? UMA's abilities to manage user-to-user sharing and user-controlled authorization servers set it apart from nearly every other protocol in the internet security space today. Although this makes UMA a fairly complex multistep protocol with many parties and moving

[1] https://docs.kantarainitiative.org/uma/rec-uma-core-v1_0.html

[2] Not to be confused with the relying party (RP) from chapter 13. The RqP is usually a person, whereas the RP is usually a computer. Yes, we know it's a little confusing.

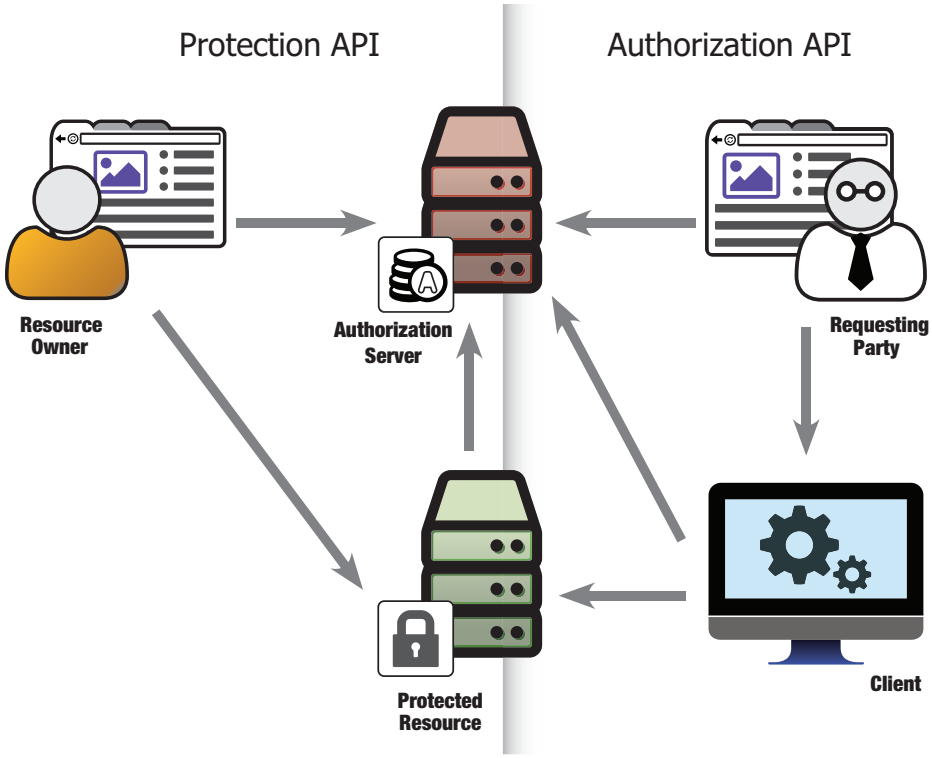

Figure 14.1 Components of the UMA protocol

parts, it also makes it a powerful protocol that can address problems unreachable by others technologies.

For a concrete look at things, let's go back to our photo-printing example. What if instead of Alice printing her own photos using a third-party service, Alice's best friend Bob wants to print some of their shared concert photos stored on Alice's account? First, Alice can tell her photo-printing service to use her own personal authorization server. This gives Alice the ability to set up a policy with her authorization server that says, in effect, "When Bob shows up, he can read any of these photos." This type of access control is relatively common, but in UMA the rights given to Bob also extend to a piece of software that Bob is running on his behalf. In this scenario, Bob has his own account at a cloud-printing service, which he points at Alice's photos. Alice's authorization server then asks Bob to prove who he is using a set of *claims*. As long as what Bob can provide matches what Alice's policy requires, Bob's printer service can gain access to the photos that Alice has shared with Bob, all without Bob needing to impersonate Alice. Bob also doesn't necessarily need to be able to log in to Alice's authorization server, and he certainly doesn't need an account at Alice's photo-sharing site.

With this setup, the resource owner can allow a requesting party to access resources even if the resource owner isn't present at the time of the request. As long as the client

can satisfy the demands in the policy of the resource in some fashion, it will be able to get a token on behalf of the requesting party. This token can be used at the resource server just like any other OAuth access token, except that now the resource server will be able to see the full chain of delegation from the resource owner to the requesting party to the client and make authorization decisions based on that.

Although UMA can work in a static world in which all parties know each other, UMA can allow components to be introduced at runtime under the guidance of authorized parties. By allowing the resource owner to bring their own authorization server, UMA sets the stage for a truly user-centric information economy, where end users have a say in not only which services can act on their behalf but also which other parties—users and software alike—can access their data.

UMA also defines a means for the resource server to register references to the resources it protects at the authorization server. These are known as *resource sets* and represent bundles of protected resources that can be attached to policies and accessed by clients. For example, Alice's photo service can register a resource set for Alice's vacation photos, another for Alice's private photos, and yet another for information about Alice's account overall. Each of these resource sets can have separate policies that let Alice determine who can use which piece of information. The resource set registration protocol is roughly analogous to the dynamic client registration protocol we covered in chapter 12. Interestingly enough, dynamic client registration can also trace some of its roots to UMA, where the problem of introducing new clients to authorization servers at runtime was more immediate.

Where OAuth pushed the boundaries of what was acceptable security policy by letting end users make runtime decisions within an ecosystem, UMA is pushing the boundaries of what is inside that ecosystem to begin with. What's allowed by policy always lags behind what is possible with technology, but UMA's capabilities are proving a powerful draw and are beginning to push the conversation forward about what kind of security is possible.

14.1.2 How the UMA protocol works

Let's take a look at a concrete UMA protocol transaction from start to finish. As you saw in chapter 6, OAuth is a protocol with many options and extensions. As a protocol built on OAuth, UMA inherits all of that optionality and adds more of its own on top of that. We could easily spend several chapters, if not a whole book, to cover it in depth.[3] Although we don't have the time or space to thoroughly explore this complex protocol, we can at least give a reasonable overview here. In this example, we're going to assume a full cold boot situation, where none of the services knows about each other ahead of time and everything needs to be introduced. We're going to be using service discovery and dynamic client registration where needed to introduce components to each other, instead of relying on manual registration. Where UMA uses more traditional OAuth

[3] If you like that idea, please contact our publisher and tell them so.

tokens, we're going to be using the authorization code flow that we covered in depth in chapter 2. We're also going to be simplifying some parts of the process to ease explanation and understanding, eliding over some details and bypassing some parts of the protocol that are underspecified or under active review by the working group. Finally, although UMA 1.0 has been finalized, the specification is still in active development by the community and many of the concrete examples here (and some of the architectural assumptions) might not apply to future versions of the protocol. With those assumptions in mind, the overall UMA dance looks like what is shown in figure 14.2.

Let's take a walk through each of the major steps in figure 14.2 in greater detail. Each paragraph is numbered to correspond with the same section of the sequence diagram there.

1. The resource owner introduces the resource server to the authorization server. The way that this happens is out of scope for the UMA protocol, but there are some provisions for how it might happen. In a tightly bound UMA ecosystem, the resource owner might be able to pick the authorization server from a list. In a more widely distributed space such as the internet, the resource owner can give the protected resource their WebFinger ID to allow discovery of their personal authorization server, much like you saw happen with identity server discovery in chapter 13. Somehow, though, the resource server ends up with a URL for the authorization server known as its *issuer* URL.

2. The resource server discovers the authorization server's configuration and registers as an OAuth client. Like OpenID Connect covered in chapter 13, UMA provides a service discovery protocol that allows other components in the system to discover vital information about the authorization server. The discovery information is hosted at a URL based on the authorization server's *issuer* URL, with `/.well-known/uma-configuration` appended to it, and its contents are a JSON document with information about the UMA authorization server.

```
{
  "version":"1.0",
  "issuer":"https://example.com",
  "pat_profiles_supported":["bearer"],
  "aat_profiles_supported":["bearer"],
  "rpt_profiles_supported":
  ["https://docs.kantarainitiative.org/uma/profiles/uma-token-bearer-1.0"],
  "pat_grant_types_supported":["authorization_code"],
  "aat_grant_types_supported":["authorization_code"],

"claim_token_profiles_supported":["https://example.com/claims/formats/
   token1"],
  "dynamic_client_endpoint":"https://as.example.com/dyn_client_reg_uri",
  "token_endpoint":"https://as.example.com/token_uri",
  "authorization_endpoint":"https://as.example.com/authz_uri",

"requesting_party_claims_endpoint":"https://as.example.com/rqp_claims_uri"  ,

"resource_set_registration_endpoint":"https://as.example.com/rs/rsrc_uri",
  "introspection_endpoint":"https://as.example.com/rs/status_uri",
  "permission_registration_endpoint":"https://as.example.com/rs/perm_uri",
  "rpt_endpoint":"https://as.example.com/client/rpt_uri"
}
```

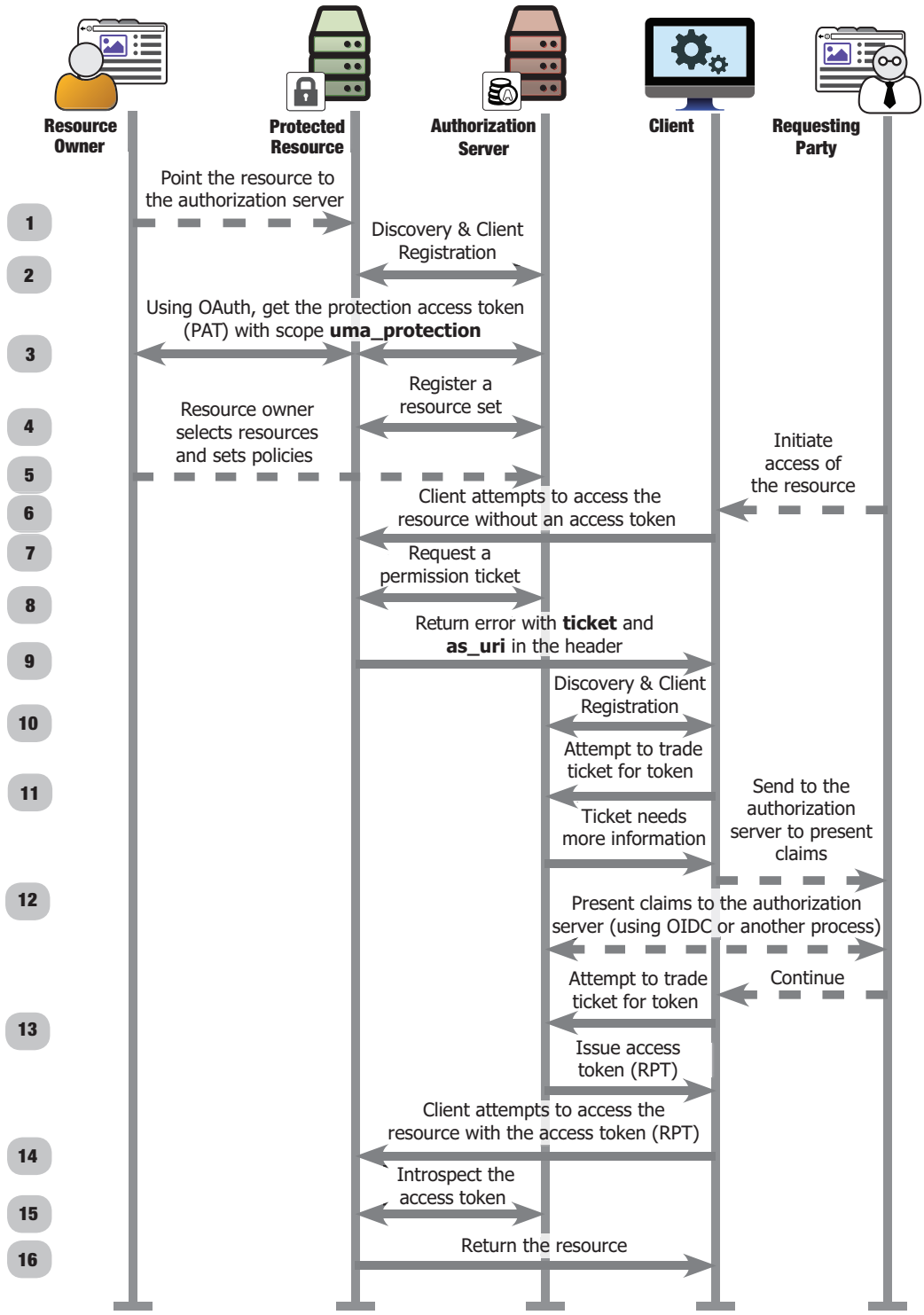

Figure 14.2 The UMA protocol in detail

This information includes things such as the authorization endpoint and token endpoint for OAuth transactions as well as UMA-specific information such as where to register resource sets, used in a future step. Notice that, like OAuth and OpenID Connect, UMA requires TLS to protect the HTTP transactions throughout the entire protocol.

The resource server can then register itself as an OAuth client using dynamic client registration, which we covered in detail in chapter 12, or register using some kind of out-of-band static process. Ultimately, this is like any other OAuth client, and the only UMA-specific aspect of this step is that the resource server needs to be able to obtain tokens with the special scope of `uma_protection`. These tokens are used to access special functionality at the authorizations server in the next steps.

3. The resource owner authorizes the resource server. Now that the resource server is acting in the capacity of an OAuth client, it needs to be authorized by the resource owner like any other OAuth client. As with OAuth, there are many options that end in the resource server getting an access token with the appropriate rights, but since this is action taken directly on behalf of the resource owner, it's common to use an interactive OAuth grant type such as the authorization code flow to accomplish this.

The access token that the resource server obtains from this process is known as the protection API token, or *PAT*. The PAT needs to have at least a scope of `uma_protection`, but it could have other scopes associated with it as well. The resource server uses the PAT to manage protected resources, request permission tickets, and introspect tokens. These actions are collectively known as the *protection API* and the authorization server provides them.

It is important, and confusing, to realize at this point that the protected resource is now acting as the OAuth client and the authorization server is acting as the protected resource through its protection API. But remember, this is not that unreasonable, since each component of the OAuth ecosystem is a role that can be played by different pieces of software at different times. The same software can easily act as both a client and protected resource, for example, depending on what the overarching API is trying to do.

4. The resource server registers its resource sets with the authorization server. The authorization server now needs to be told about the resources that the resource server is protecting on behalf of the resource owner. The resource server registers its resource sets in a protocol analogous to the dynamic client registration protocol. Using the PAT as authorization, the resource server sends an HTTP POST message with details about each of the resource sets that it wants to protect to the resource set registration URI.

```
POST /rs/resource_set HTTP/1.1
Content-Type: application/json
Authorization: Bearer MHg3OUZEQkZBMjcx
```

```
{
  "name" : "Tweedl Social Service",
  "icon_uri" : "http://www.example.com/icons/sharesocial.png",
  "scopes" : [
    "read-public",
    "post-updates",
    "read-private",
    "http://www.example.com/scopes/all"
  ],
  "type" : "http://www.example.com/rsets/socialstream/140-compatible"
}
```

These details include a display name, an icon, and most important a set of OAuth scopes that are associated with the resource set. The authorization server assigns a unique identifier to the resource set and returns it along with a URL where the resource server can send the resource owner to interactively manage policies associated with this resource set.

```
HTTP/1.1 201 Created
Content-Type: application/json
Location: /rs/resource_set/12345

{
  "_id" : "12345",
  "user_access_policy_uri" : "http://as.example.com/rs/222/resource/333/
   policy"
}
```

The Location header contains a URL for managing the resource set registration itself using a RESTful API pattern. By using HTTP `GET`, `PUT`, and `DELETE` verbs along with the `POST` verb, the resource server can read, update, and remove its resource sets, respectively.

5. The resource owner sets policies pertaining to the resource sets with the authorization server. The resource set is now registered, but no one has said anything about how it can be accessed. Before any client can request access to the resource set, the resource owner needs to set policies on the resources to indicate who can access the resource and under what circumstances. The nature of these policies is completely out of scope for UMA, as there are nearly infinite ways that a policy engine could be written and configured. Common options could include date ranges, user identifiers, or limits on the number of times a resource can be accessed.

Each of these policies can also be associated to a subset of the scopes possible for each resource set, offering the resource owner a rich way to express their sharing intentions. For example, the resource owner could decide that any user with an email address from their family's domain can read all of their photos, but only certain identified individuals would be able to upload new photos.

In the end, it comes down to the requesting party and their client being able to present a set of claims that fulfill the requirements of the policies that have been configured. Importantly, a resource set with no policy on it needs to be considered inaccessible until an appropriate policy has been configured. This restriction prevents a naïve

authorization server failing in an open state, where their resource would be made inadvertently available to anyone who asked. After all, if I have a resource with no required claims set to it, doesn't that mean I can fulfill all of the policies and get a token by presenting no claims at all?[4]

Once the policies have been set up, the resource owner generally exits the picture and the requesting party begins their half of the UMA dance. It's possible that the authorization server has an advanced runtime policy engine that allows the resource owner to be prompted for authorization when someone else, the requesting party, attempts to access their resource. We're not showing that mechanism here, however.

6. The requesting party directs the client to access the resource server. This step is analogous to the start of a normal OAuth transaction in which the resource owner instructs their client to access something on their behalf. As in OAuth, the means by which the client learns the URL of the protected resource or the knowledge required to access the API protected there is outside the scope of the specification. Unlike in OAuth, though, the requesting party is directing the client application to access a resource controlled by someone else, and the client might not know where the associated authorization server is.

7. The client requests the protected resource. The client starts the process by making a request without sufficient authorization for the resource. Most commonly, this is going to be a request with no access token.

```
GET /album/photo.jpg HTTP/1.1
Host: photoz.example.com
```

When we discussed different scope patterns in chapter 4, we saw that OAuth can be used to protect many different styles of API because the access token gives additional context to the request, such as an identifier for the resource owner or the scopes associated with it. This lets the protected resource serve different information depending on the data associated with the access token, such as serving different user information from a single URL or serving subsets of information based on the scopes associated with the token and the user who authorized the token. In OpenID Connect, this feature of OAuth allows the UserInfo endpoint to be a single URL to serve all identities on the server without leaking user identifiers to the client ahead of time, as we saw in chapter 13. In UMA, the resource server needs to be able to tell from the context of this initial HTTP request which resource set the client's request is attempting to access, and therefore which resource owner and which authorization server are being represented. We don't have an access token to help us make this decision, so we can only look at the URL, headers, and other parts of the HTTP request. This restriction effectively limits the types of APIs that UMA can be used to protect to those that differentiate resources based on URLs and other HTTP information.

8. The resource server requests a permission ticket from the authorization server to represent the requested access and hands that ticket to the client. Once the resource

[4] Yes, this was a real bug in a real implementation. No, I don't want to talk about it.

server knows which resource set the request is attempting to access, and therefore which authorization server is associated with that resource set, the resource server requests a permission ticket to represent the access request by sending an HTTP POST message to the authorization server's permission ticket registration endpoint. This request contains the identifier of the resource set as well as a set of scopes that the resource server thinks would be appropriate for access, and is authorized by the PAT. The scopes in this request can be a subset of those on the resource set, allowing a resource server to potentially limit the client's access as it sees fit. Of course, the client could be capable of more actions than what's obvious from this initial request, but the resource server has no way of guessing that a priori.

```
POST /tickets HTTP/1.1
Content-Type: application/json
Host: as.example.com
Authorization: Bearer 204c69636b6c69

{
  "resource_set_id": "112210f47de98100",
  "scopes": [
      "http://photoz.example.com/dev/actions/view",
      "http://photoz.example.com/dev/actions/all"
  ]
}
```

The authorization server makes sure that the PAT represents the same resource server that registered the resource set in the first place, and that the scopes requested are all available on that resource set. The authorization server then creates and issues a permission ticket, sending it back to the resource server as a string in a simple JSON object.

```
HTTP/1.1 201 Created
Content-Type: application/json

{
"ticket": "016f84e8-f9b9-11e0-bd6f-0021cc6004de"
}
```

The resource server doesn't have to keep any reference to these tickets or manage them, as they're handles for the client to interact with the authorization server throughout the UMA process. The authorization server will automatically expire and revoke them as necessary.

9. The resource server returns the ticket to the client along with a pointer to the authorization server. Once it has the ticket, the resource server can finally respond to the client's request. The resource server uses the special `WWW-Authenticate: UMA` header to give the client the ticket as well as the issuer URL of the authorization server protecting this resource.

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: UMA realm="example",
  as_uri="https://as.example.com",
  ticket="016f84e8-f9b9-11e0-bd6f-0021cc6004de"
```

The only part of this response dictated by the UMA protocol is the header, and the rest of the response, including the status code, body, and other headers, are up to the protected resource. In this way, a resource server is free to serve publicly available information alongside the indicator telling the client how to obtain a higher level of access. Or, if the client did present an access token in its initial request but the token did not have the full set of available access rights associated with it, the resource server can serve content appropriate for the level of access that the client presented while indicating that the client could attempt to step up its access. In our example, the client sent no token and no public information was available on the API, so the server returned an HTTP 401 code with the header.

10. The client discovers the authorization server's configuration and registers with it. As with the resource server, the client needs to find out where the authorization server is and how to interact with it during the next steps. Since the process here is parallel, we won't copy its details. At the end of the process, the client has its own set of credentials that it can use to interact with the authorization server, distinct from those used by the protected resource.

Do I need a token to get a token?

In the 1.0 version of UMA, the client also needs to get an OAuth access token known as the *authorization access token*, or AAT. The intent of this token is to bind the requesting party to the client and the authorization server, in much the same way the PAT functions on the other side of the system. However, since the RqP can be required to present claims interactively in a future step of the process, this binding is not strictly necessary. Furthermore, in order to authorize an AAT, the RqP needs to be able to log in to the authorization server and authorize a client to get a token with a special scope, `uma_authorization`. However, the RqP isn't guaranteed to have any relationship with the authorization server, only the resource owner is, and therefore the RqP can't be expected to be able to pursue normal OAuth transactions. For these and other reasons, future versions of the UMA protocol may do away with the AAT and we've minimized its importance in our discussion. Future versions of the UMA protocol may use different mechanisms to represent and carry the RqP's consent in part of the process.

11. The client presents the ticket to the authorization server to get an access token. This process is analogous to a client presenting an authorization code in the authorization code grant type, but using the ticket gathered from the resource server as the temporary limited credential. The client sends an HTTP POST message to the authorization server that includes the ticket as a parameter.

```
POST /rpt_uri HTTP/1.1
Host: as.example.com
Authorization: Bearer jwfLG53^sad$#f

{
"ticket": "016f84e8-f9b9-11e0-bd6f-0021cc6004de"
}
```

The authorization server examines the ticket to find out which resource set the request is associated with. Once the resource set is known, the authorization server can determine the policies associated with the resource set, and therefore which claims the client is going to need to present in order to obtain the access token. Since the ticket has just been created in our example, the authorization server determines that it doesn't have sufficient claims associated with it in order to fulfill the policies. The authorization server sends back an error response to the client indicating that the client needs to gather some claims and prove to the authorization server that both the requesting party and the client itself should be allowed access.

```
HTTP/1.1 403 Forbidden
Content-Type: application/json
Cache-Control: no-store

{
 "error": "need_info",
 "error_details": {
   "authentication_context": {
     "required_acr": ["https://example.com/acrs/LOA3.14159"]
   },
   "requesting_party_claims": {
     "required_claims": [
       {
         "name": "email23423453ou453",
         "friendly_name": "email",
         "claim_type": "urn:oid:0.9.2342.19200300.100.1.3",
         "claim_token_format":
["http://openid.net/specs/openid-connect-core-1_0.html#HybridIDToken"],
         "issuer": ["https://example.com/idp"]
       }
     ],
     "redirect_user": true,
     "ticket": "016f84e8-f9b9-11e0-bd6f-0021cc6004de"
   }
 }
}
```

This example response contains a set of hints about the kinds of claims the client might want and where to gather them, in this case OpenID Connect claims from the given OpenID Connect issuer.

12. The client gathers claims and submits them to the authorization server. At this stage, the client can do a few different things to get the authorization server what it's asking for. The details of the claims-gathering process are left intentionally vague by the UMA protocol in order to allow for a wide variety of situations and circumstances to use it.

If the client has the claims already, and in a format verifiable by the authorization server, then it can send them directly in another request for the token.

```
POST /rpt_authorization HTTP/1.1
Host: www.example.com
Authorization: Bearer jwfLG53^sad$#f
```

```
{
    "rpt": "sbjsbhs(/SSJHBSUSSJHVhjsgvhsgvshgsv",
    "ticket": "016f84e8-f9b9-11e0-bd6f-0021cc6004de",
    "claim_tokens": [
      {
        "format":
"http://openid.net/specs/openid-connect-core-1_0.html#HybridIDToken",
        "token": "..."
      }
    ]
}
```

This approach works well when the client is presenting claims about itself or the organization that has deployed the client software. An authoritative party can sign these kinds of claims so that the authorization server can verify and validate them directly. It's less helpful if the client needs to submit information about the requesting party, since the relationship between the requesting party and the client is undefined, though even this is possible if there's a strong trust relationship between the client and the authorization server.

If instead the client needs to have the requesting party submit claims, such as their identity, then the client redirects the requesting party to the claims-gathering endpoint of the authorization server. The client includes its own client ID, the ticket value, and a URI to redirect to after the claims have been gathered.

```
HTTP/1.2 302 Found
Location: https://as.example.com/rqp_claims?client_id=some_client_id&state=abc
&claims_redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fredirect_
claims&ticket=016f84e8-f9b9-11e0-bd6f-0021cc6004de
```

At this endpoint, the requesting party can interact with the authorization server directly to provide the required claims. This process is once again left open to interpretation in the UMA specification, but in our example the requesting party is going to log in to the authorization server using their OpenID Connect account. The UMA authorization server is now acting as an OpenID Connect relying party,[5] giving the authorization server access to the requesting party's identity information, which can be used to fulfill the policy requests.

When the authorization server is satisfied by the claims-gathering process, it redirects the requesting party back to the client to signal a continuation of the process.

```
HTTP/1.1 302 Found
Location: https://client.example.com/redirect_claims?&state=abc
&authorization_state=claims_submitted
```

This process uses front-channel communication between the client and the authorization server, as with the authorization endpoint in regular OAuth discussed in chapter 2. However, the redirect URI used here is distinct from that used for the authorization code or implicit grant types.

[5] This means that our UMA authorization server is now variously playing the role of an OAuth authorization server, protected resource, and client in support of this crazy ride.

Whichever process is used to submit the claims, the authorization server associates the claims with the ticket. The client still needs to submit the ticket again to get a token.

13. The client presents the ticket again and tries to get a token. This time it works since the ticket now has associated with it a set of claims that satisfy the policies on the resource set. These policies also map to a subset of scopes, allowing the authorization server to determine the final access rights of the resulting token. The authorization server issues the token back to the client in a JSON document, analogous to the return from OAuth's token endpoint.

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "rpt": "sbjsbhs(/SSJHBSUSSJHVhjsgvhsgvshgsv"
}
```

Finally, the client has its access token and can try getting that resource again. As in OAuth, the content and format of the token itself is completely opaque to the client.

14. The client presents the access token to the resource server. The client once again calls the protected resource, but this time it includes the token it just received from the authorization server.

```
GET /album/photo.jpg HTTP/1.1
Host: photoz.example.com
Authorization: Bearer sbjsbhs(/SSJHBSUSSJHVhjsgvhsgvshgsv
```

This request is an absolutely standard OAuth bearer token request, and there's nothing UMA specific about it. The client had to do a few special things to get to this point, but now it can act like any other OAuth client.

15. The protected resource determines what the token is good for. Now that the protected resource has received the token from the client, the resource server needs to figure out whether the token presented is good for what the client is trying to do. However, by the design of the UMA protocol, the resource server is separated from the authorization server, which makes any kind of local lookup of the token information impossible to count on.

Thankfully for us, we've already covered the two most common methods of connecting a protected resource to an authorization server in chapter 11: JSON Web Tokens (JWT) and token introspection. Since UMA is a network-based protocol and the authorization server is expected to be online at runtime to respond to network queries, it is much more common to use token introspection in this step, so we'll look at that here. The request from the resource server is identical to those that we've already examined, except that instead of using its client credentials it authorizes the call using the PAT. The response from the server is slightly different, as UMA extends the introspection response data structure with a `permissions` object that contains detailed information about the permissions that have been fulfilled in order to issue the token.

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

   {
    "active": true,
    "exp": 1256953732,
    "iat": 1256912345,
    "permissions": [
      {
        "resource_set_id": "112210f47de98100",
        "scopes": [
          "http://photoz.example.com/dev/actions/view",
          "http://photoz.example.com/dev/actions/all"
         ],
        "exp" : 1256953732
      }
    ]
   }
```

The token itself could be good for multiple resource sets and multiple permissions sets, depending on the setup of the policy engine on the authorization server. The resource server needs to determine whether the token has the correct rights associated with it to fulfill the request at hand. As with OAuth, it's entirely up to the resource server to decide whether the token is good enough. If the token doesn't have the correct rights, the resource server can repeat the process of registering a permission ticket and handing it back to the client to start the process over again. A client responding to such an error message would behave as it had previously and fetch a new token.

16. Finally, the resource is returned to the client. As in OAuth, once the protected resource is satisfied by the incoming token, it returns the response appropriate to the API that it has been protecting.

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "f_number": "f/5.6",
  "exposure": "1/320",
  "focal_length_mm": 150,
  "iso": 400,
  "flash": false
}
```

This response can be any HTTP response, and it could also include another `WWW-Authenticate: UMA` header to indicate that if the client wants to get additional access, it can attempt to do so.

Throughout the entire process, neither the resource owner's credentials nor the requesting party's credentials were revealed to the resource server or the client. Additionally, sensitive personal information about either party has not been disclosed between them. The requesting party need only prove whatever minimum amount is required to fulfill the policies set by the resource owner.

14.2 Health Relationship Trust (HEART)

As we've seen throughout this book, and as we're sure you've seen in the real world, OAuth can be used to protect a wide variety of protocols and systems. However, the large amount of flexibility and optionality makes it difficult to ensure interoperability and compatibility across different deployments. As discussed in chapter 2, this tends not to be a problem when dealing with different APIs and providers. However, if you're working in a single sector, such as healthcare, it's likely you'll be working with a common set of APIs. Here, it's more valuable to have a limited set of good options and clear guidance for deployments. This way, clients from one vendor can work out of the box with authorization servers from another vendor and protected resources from yet a third.

The OpenID Foundation's Health Relationship Trust[6] (HEART) working group was founded in 2015[7] to address the particular needs of the electronic health system community. The goal of the working group is to provide profiles of existing technology standards that are suitable for healthcare use cases while, wherever possible, retaining compatibility with widely available standards with general applicability. The HEART working group is building on top of OAuth, OpenID Connect, and UMA. By locking down optional features and codifying best practices, HEART seeks to promote both higher security and greater interoperability between independent implementations.

14.2.1 Why HEART matters to you

The HEART profiles are among the first of a set of standards that increase both security and interoperability within a given sector, in this case healthcare. This type of profiling is likely to become increasingly common as more industries move to an API-first ecosystem, and in the future you might be required to ensure that something is "HEART-compliant" in addition to being able to implement OAuth properly.

In contrast to many past efforts of digitizing aspects of healthcare, HEART explicitly places decision capability and control in the hands of end users, particularly patients and their healthcare providers. Instead of centralizing data, control, and security decisions, HEART envisions and creates a landscape in which data can be securely distributed and connected at the behest of the producers and consumers of the data. A patient should be able to bring their own application and connect to their health records, whether or not their healthcare provider and the application developer have ever heard of each other. As health data is extremely personal and sensitive, security is of the utmost importance.

In order to accomplish this, HEART defines a set of technical profiles that raise the baseline of both security and interoperability between components in an OAuth ecosystem. These profiles are being driven by use cases and requirements that are

[6] http://openid.net/wg/heart/

[7] One of your authors was a founding member of the working group.

potentially educational to all you OAuth practitioners reading this book. And if you're a member of a health IT team, you should *really* be paying close attention there.

14.2.2 The HEART specifications

A suite of specifications defines HEART's ecosystem, each specification covering a different part of the technology stack. These specifications are a *conformant subset* of the protocols they profile, meaning they don't allow or require anything illegal in the underlying protocols, but in many cases they require components that were previously optional or add new optional functionality using existing extension points in a compatible fashion. In other words, a HEART-compliant OAuth client is also a fully compliant OAuth client, but a general OAuth client might not support all the features and options required by HEART.

Now, it's entirely probable that many of the readers of this book will never call or deploy a healthcare-related API. As a consequence, you might be thinking, "So what?" There are two ideas to take away here. First, HEART wraps the standard API and the security technology together in such a way that they're immediately usable together, regardless of the source implementation. This is an important pattern to observe regardless of the sector in which the technology is found. Second, many of the profiling decisions made in HEART are useful outside of the healthcare space.

To accommodate this, HEART is built along two distinct dimensions: mechanics and semantics. This division is intended to allow HEART to have a reach far outside the healthcare community while still maintaining direct applicability within it. We'll cover each of these dimensions and the associated specifications in the next sections.

14.2.3 HEART mechanical profiles

The three mechanical specifications are built on top of OAuth, OpenID Connect, and UMA. These three profiles aren't specific to any particular kind of API, and they're in no way healthcare specific. Consequently, these profiles can be used in a variety of circumstances in which higher security and interoperability are desired. The mechanical profiles also build on each other, much the same way that the protocols they're profiling build on each other: the OpenID Connect profile inherits directly from the OAuth profile, whereas the UMA profile inherits from both the OAuth profile and the OpenID Connect profile.

The *HEART profile of OAuth* differs from core OAuth in several ways. First, since the profile doesn't need to fit as wide a set of use cases as does OAuth itself, it can contain clear guidance on which kinds of OAuth grant types should be used for which kinds of clients. For instance, only in-browser clients are allowed to use the implicit grant type, whereas only back-channel server applications dealing with bulk operations are allowed to use the client credentials grant type. HEART notably removes client secrets from all clients and instead requires all clients, regardless of grant type, to register a public key with the authorization server. This key is used by

any clients that authenticate to the token endpoint (using the authorization code or client credentials grant types) and is available for any other protocol uses. These decisions increase the baseline security of the entire ecosystem at the cost of slightly increased complexity for all parties. However, the keys and their use aren't unique to HEART: the key format is JOSE's JWK as discussed in chapter 11 and the JWT-based authentication is defined by OpenID Connect, as we briefly mentioned in chapter 13.

HEART OAuth authorization servers are also required to support both token introspection and token revocation (we covered both of these in chapter 11), as well as provide a standardized service discovery endpoint (based on the one you saw in chapter 13). All tokens that are issued from a HEART OAuth authorization server are asymmetrically signed JWTs (which you saw in chapter 11) with required claims and lifetimes specified by the profile. The HEART profile also requires that dynamic client registration (covered in chapter 12) be available at the authorization server, though of course clients can still be registered manually or use software statements. This set of capabilities allows clients and protected resources to depend on core functions being available across a wide array of implementations, allowing for true out-of-the-box interoperability.

HEART OAuth clients are required to always use the state parameter with a minimum amount of entropy, which immediately closes a number of session fixation attacks as discussed in chapter 7. Clients are also required to register their full redirect URIs, which are compared using exact string matching at the authorization server, as discussed in chapter 9. These requirements codify best practices to increase security and give developers a set of capabilities to depend on.

The HEART profile of OpenID Connect explicitly inherits all of the requirements and features of the OAuth profile, which helps keep the delta to implement OIDC on top of OAuth small. In addition, the HEART OIDC profile requires the identity provider (IdP) to always sign ID tokens asymmetrically as well as offer the output of the UserInfo endpoint as an asymmetrically signed JWT in addition to the unsigned JSON used by default in OIDC. Since all clients are required to register their own keys, the IdP is also required to offer optional encryption for all of these JWTs. The IdP has to be able to take requests using the OIDC request object, using the client's keys to validate the request.

The HEART profile of UMA selects specific components from UMA's potential extension points while inheriting from the other two mechanical HEART profiles. For example, all of the RPTs and PATs inherit the HEART requirements for OAuth access tokens, and are therefore signed JWTs that are also available for token introspection. Authorization servers are required to support gathering of requesting party claims through the use of interactive OpenID Connect login, which is itself conformant with the HEART OIDC profile. The HEART profile also mandates that dynamic resource registration be available at the authorization server in addition to dynamic client registration.

14.2.4 HEART semantic profiles

The two semantic profiles are healthcare specific and focus on the use of the Fast Healthcare Interoperable Resources (FHIR)[8] specifications. FHIR defines a RESTful API for sharing healthcare data, and HEART's semantic profiles are designed to secure it in a predictable manner.

The HEART profile of OAuth for FHIR defines a standard set of scopes that are used to provide differential access to FHIR resources. The HEART profile divides the scopes of access by type of resource and general access target. This allows protected resources to determine the rights associated with an access token in a predictable manner, cleanly mapping HEART scope values to medical record information.

The HEART profile of UMA for FHIR defines a standard set of claims and permission scopes that can be used across different kinds of FHIR resources. These scopes are specific to individual resources and can provide guidance to policy engines on how to enforce them. HEART also defines in greater detail specific claims for users, organizations, and software, as well as how these claims can be used to request and grant access to protected resources.

14.3 International Government Assurance (iGov)

Just as HEART profiles security protocols for the healthcare sector, the International Government Assurance (iGov) working group of the OpenID Foundation[9] seeks to define a set of profiles for use in government systems. The focus of iGov is largely on how federated identity systems, such as OpenID Connect, can be used to allow citizens and employees to interact with government systems.

14.3.1 Why iGov matters to you

The iGov profiles will be built on OpenID Connect, which is of course built right on top of OAuth. The requirements set forth in these profiles will affect a large number of government systems and the systems that connect to them through these standard protocols. Governments have traditionally moved very slowly and lagged behind industry in their adoption of new technology; governments are big enterprises, reluctant to change and cautious about taking risks. This makes their systems particularly sticky: once something is put into practice in a government system, it's likely to be there for a long time. Even years from now, when OAuth 2.0 is a distant memory for most of the internet and we've all forgotten what the big deal about JSON and REST was, there's a good chance that a government system will still be actively using this as a legacy protocol and require interface and upkeep.

Governments also have a way of being large enough and important enough that whatever solution they decide on tends to set requirements for many others in unrelated spaces. Often, governments use this dominance and sway to dictate a requirement

[8] https://www.hl7.org/fhir/

[9] One of your authors is also a founding member of this group. Small world, right?

and have everyone react to it. The iGov approach is different, so far, in that the government stakeholders involved aren't trying to define the technology being used but rather how it can be used for their use cases. This means that a keen OAuth developer will be able to watch this effort and learn about a special set of constraints and means around them.

Finally, as with HEART, the core components of iGov are planned to be generally applicable outside of the government sector. In fact, the iGov working group is starting with the HEART mechanical specifications for OAuth and OpenID Connect as its base. As such, it's possible that nongovernment systems will start offering iGov compliance and HEART compliance as features, because doing so would allow them to interact with both the constrained profiles as well as the rest of the general OAuth ecosystem. You may see these as requirements to your systems in the future, or some similar profiles that could be based on this work.

14.3.2 The future of iGov

The iGov working group is just getting started as of this book's publication, but there are already key stakeholders from major governments around the world. There's a lot that's still unknown about iGov, including whether it will be successfully built and widely adopted. However, it's going to be an important space to watch for the reasons listed here, and the OAuth practitioners reading this book could learn a lot from this effort. And if you work in the government and citizen identity space, it will likely be a good idea to get involved yourself.

14.4 Summary

OAuth is a great foundation for building new protocols.

- UMA allows resource servers and authorization servers to be introduced together in a way that can be highly dynamic and user-driven across security domains.
- UMA adds new parties to the OAuth dance, most notably the requesting party, allowing true user-to-user sharing and delegation.
- HEART applies several open standards based on OAuth to the healthcare domain, profiling them to increase security and interoperability.
- HEART defines both mechanical and semantic profiles, allowing the lessons learned to reach beyond the healthcare domain and find wide applicability.
- iGov is in the formative stages of development, but it will define a set of profiles for government identity systems that could have far-reaching consequences.

We've been able to do a lot of things with OAuth and simple bearer tokens, but what if there were another option? In the next chapter, we'll look at a snapshot of the ongoing work of Proof of Possession tokens.